Architecture Decision Document

This document builds collaboratively through step-by-step discovery. Sections are appended as we work through each architectural decision together.

Project Context Analysis

Requirements Overview

Functional Requirements:

- Identity & access: Entra ID (EasyAuth) sign-in, minimal profile capture, and user-only data access.
- Programme participation: join/leave/pause with single-programme MVP; opt-in only and no penalties.
- Matching: scheduled cadence, random/lightly constrained matching, avoid repeats, odd-participant handling, idempotent cycles.
- Notifications: match emails with first-move prompt and Teams deep link; email is authoritative fallback.
- Privacy & trust: no individual-level reporting; strict deletion rights; transparency page; event logging of system actions only.
- Administration: role assignment, programme lifecycle controls, aggregate programme health only.

Non-Functional Requirements:

- Security & privacy: TLS, encryption at rest, least-privilege access, no surveillance analytics, GDPR/UK GDPR/CCPA compliance.
- Reliability: predictable matching execution, atomic runs, clear failure behaviour, idempotent retries.
- Performance: basic web performance expectations (sub-3s load), no real-time constraints.
- Accessibility: WCAG 2.1 AA aspiration, keyboard support, readable contrast.
- Observability: structured logs, error monitoring, aggregate-only insights.
- Email delivery: retry logic, delivery logging, within-60-minute window post-match.

Scale & Complexity:

- Primary domain: SPA + serverless API + scheduled jobs
- Complexity level: Medium (trust/privacy constraints dominate)
- Estimated architectural components: 6-8 (SPA, auth, API, matching worker, data store, notification service, logging/monitoring, CI/CD)

Technical Constraints & Dependencies

- Azure Static Web Apps + EasyAuth (no custom auth flows in MVP).
- Azure Functions v4 (.NET isolated) with explicit auth checks per handler.
- Azure Table Storage partitioning by ProgrammeId; avoid cross-partition scans.
- Idempotent matching runs keyed by ProgrammeId + CycleId.
- OpenAPI.NET generated specs aligned to DTOs and contract tests.
- Email via Azure Communication Services; Teams is secondary.
- Single-tenant MVP with future multi-tenant readiness.
- React 18 + Vite 5 SPA; minimal routing; no state management library in MVP.

Cross-Cutting Concerns Identified

- Trust contract enforcement (anti-surveillance, opt-in only, no individual metrics).
- Authorisation at every boundary with untrusted claims validation.
- Data minimisation, retention, and deletion workflows.
- Idempotency and failure handling for scheduled matching.
- Aggregate-only reporting constraints for admin/owner views.
- UX tone/clarity as a product promise (low-pressure, low-effort).

Failure Risks & Early Preventive Guards

- **Trust breach perception:** any hint of surveillance or manager visibility kills adoption -> enforce aggregate-only endpoints and guard-suite tests.
- **Cross-programme data leakage:** partition mistakes expose data -> repository-level key enforcement and negative contract tests.
- **Matching integrity failure:** duplicate runs or partial matches undermine credibility -> idempotency keys and atomic cycle processing.
- **Notification reliability gap:** silent delivery failures feel like "ghosting" -> delivery logging and retry strategy.
- **UX drift into "programme" feel:** extra screens or metrics create obligation -> single-screen core and consistent "no pressure" copy.

Architectural Drivers

- **Trust by design:** privacy posture drives data minimisation and transparency surfaces.
- **Boundary discipline:** auth/authorisation and partition keys must enforce programme isolation on every query and endpoint.
- **Predictable cadence:** scheduling and idempotent matching are central reliability anchors.
- **Minimal surface area:** few routes and minimal UI states reduce risk and effort.
- **Operational clarity:** logging must explain what the system did, not what users did.

Stakeholder Lens

- **Participants:** need effortless opt-in/out, zero penalty for silence, and visible proof of non-surveillance -> minimal data capture and clear "what we store" surface.
- **Programme Owners:** need hands-off operation and credible aggregate signals -> aggregate metrics only and reliable cycle completion reporting.
- **Executive Sponsors:** need narrative evidence without privacy risk -> anonymised aggregates and story capture outside the system.
- **Security/Privacy:** require lawful basis, minimisation, retention controls, and auditability -> explicit retention workflows and DPIA-ready documentation.
- **Engineering/Ops:** need low-complexity, debuggable flows -> serverless, deterministic jobs, strong contract tests.

Cross-Functional Trade-offs

- **PM:** wants validation speed and minimal scope -> architecture stays small, low-config, no platform sprawl.

- **Engineering:** wants reliability and debuggability -> deterministic jobs, explicit failure logging, contract-first APIs.
- **UX:** wants calm, low-pressure surfaces -> avoid extra flows, keep copy consistent, no KPI-styled UI.
- **Privacy/Legal:** wants defensible minimisation and transparency -> strict retention, no behavioural analytics, clear data-handling docs.

Security & Privacy Lens

- **Attacker view:** highest value targets are identity claims, cross-programme access, and match history -> boundary validation and partition isolation are critical.
- **Defender view:** rely on SWA/EasyAuth for identity, but treat claims as untrusted per request -> explicit auth checks and schema validation everywhere.
- **Auditor view:** needs evidence of minimisation, retention, and deletion controls -> document retention policies and ensure logs exclude user behaviour.

Expert Panel Notes

- **Serverless architect:** keep Functions thin, avoid framework-building, isolate matching logic into a testable service with idempotency guarantees.
- **Data/privacy specialist:** minimise stored attributes (name, email, department, location), enforce retention windows, prefer anonymised aggregates.
- **Test architect:** prioritise contract tests for auth edge cases, cross-programme access, and idempotent matching; guard tests for anti-surveillance promises.

Component Failure Modes

- **Auth boundary:** missing/forged claims -> requests must fail closed with explicit 401/403.
- **Programme scoping:** incorrect PartitionKey usage -> accidental cross-programme data access.
- **Matching cycle:** re-run creates duplicate matches -> must detect and no-op on duplicates.
- **Notification delivery:** transient email failures -> retry with observable failure logs.
- **Retention/deletion:** stale personal data persists -> explicit delete workflows and retention checks.
- **Aggregate reporting:** small cohort sizes enable re-identification -> enforce minimum-N thresholds.
- **UX trust drift:** any copy/UI implying obligation or monitoring -> treat as defect, block release.

Clarifications & Defaults (from stakeholder decisions)

- **Trust contract visibility:** "What we store" is a primary surface on join and first-match screens; full detail behind a link.
- **Cadence promise:** best-effort within a consistent window (e.g., first Tuesday by 10:00 local time); internal alerts on delay, calm external messaging.
- **Aggregate threshold:** minimum-N fixed at 5 for MVP.
- **Retention:** match records retained 12 months; user deletion requests honoured promptly; when leaving, stop matching and retain only minimal non-identifying audit data.
- **Notifications:** email is authoritative; Teams is additive; no SMS.
- **Identity key:** Entra Object ID is canonical; email stored for display/contact only.
- **Time zones:** cadence runs per programme time zone (single-programme MVP).
- **Idempotency key:** ProgrammeId + CycleDate (programme time zone).

- **Opt-out semantics:** pause for one cycle and leave programme are both supported; pause is a per-cycle flag.
- **No follow-ups:** ignored matches receive no reminders.

Starter Template Evaluation

Primary Technology Domain

Monorepo with three components:

- Web SPA (React/Vite)
- Serverless API (Azure Functions .NET isolated)
- Marketing site (11ty)

Starter Options Considered

1. **Official Vite React TypeScript template** (for the web app)

- Command (official):

```
pnpm create vite
```

Then select React + TypeScript in prompts, or specify `--template react-ts`.

2. **Azure Functions Core Tools init** (for the API)

- Command:

```
func init collabolatte-api --worker-runtime dotnet-isolated
```

3. **Eleventy (11ty) local install** (for the marketing site)

- Commands (official):

```
pnpm init
pnpm install @11ty/eleventy
npx @11ty/eleventy
```

Selected Starter: Official Vite + Functions Core Tools + 11ty

Rationale for Selection:

- Minimal and predictable, aligns with the trust-first, low surface area ethos.
- Keeps UI and content decisions in our control.
- Monorepo enables shared theming assets for both app and marketing site.

Monorepo Structure (proposed)

```
/apps
  /web           # React + Vite SPA
  /api           # Azure Functions .NET isolated
  /marketing     # 11ty site
/packages
  /theme         # shared design tokens + CSS variables + MUI theme export
```

Workspaces (pnpm)

- Use `pnpm-workspace.yaml` at the repo root to define workspace packages.

Shared Theming (lightweight tokens)

- `/packages/theme/tokens.css` for CSS variables used by 11ty
- `/packages/theme/muiTheme.ts` exporting the MUI theme built from the same tokens

Tooling Additions (confirmed)

MUI (web app):

```
pnpm add @mui/material @emotion/react @emotion/styled
```

Playwright (E2E):

```
npm init playwright@latest
```

Storybook (component workshop):

```
npm create storybook@latest
```

Deployment (Azure Static Web Apps)

- **Two separate SWA resources**:
  - `app.collabolatte.co.uk` -> SPA + API (SWA with `app_location` + `api_location`)
  - `www.collabolatte.co.uk` -> marketing site (SWA with `app_location` only)
- **Plan:** Both SWAs on Free tier for MVP (cost-minimising), with upgrade path to Standard if/when needed.
- **Workflow build configuration**:

    - Use `app_location`, `api_location`, and `output_location` in the SWA workflow YAML to match each app folder.
    - `api_location` points to the Functions folder for the app SWA.

- **Routing/auth configuration**:

    - Use `staticwebapp.config.json`; `routes.json` is deprecated.
    - Place `staticwebapp.config.json` under the `app_location`, and ensure the build outputs it to the root of `output_location`.

Architectural Decisions Provided by Starter

Language & Runtime:

- React + TypeScript via Vite scaffold.
- Azure Functions .NET isolated worker model for the API.
- 11ty for the marketing site.

Styling Solution:

- MUI + Emotion in the web app.
- Shared CSS variables/tokens package for 11ty and MUI.

Testing Framework:

- Playwright for E2E.

Component Workshop:

- Storybook for isolated UI development.

Code Organisation:

- pnpm workspaces with `/apps` and `/packages` to share theming across SPA and marketing site.

Development Experience:

- Vite dev server for the SPA.
- 11ty local build via npx.
- Storybook for UI states and docs.

Cross-Functional Trade-offs (Starter & Monorepo)

- **PM:** wants speed to validate and clean separation between app and marketing -> two SWAs keep messaging and product surfaces distinct.
- **Engineering:** prefers minimal, predictable scaffolding -> official Vite + Functions init reduces surprises.
- **UX:** wants consistent brand across app and marketing -> shared tokens package is essential.
- **Privacy/Legal:** prefers clear boundary between marketing and authenticated app data -> separate SWAs and domains reduce accidental data coupling.

Dependency Map (Monorepo & Deployment)

- **Shared theme package** feeds both `/apps/web` (MUI theme) and `/apps/marketing` (tokens.css).

- **SWA app+API** depends on repo layout alignment (`app_location`, `api_location`, `output_location`).
- **Marketing SWA** depends on 11ty build output location matching its SWA workflow.
- **Playwright** targets `/apps/web` (app domain), not marketing; marketing can use simple link checks.

Critical Risks & Mitigations (Starter & Deployment)

- **Monorepo complexity creep:** multiple apps can drift -> keep shared tooling minimal and document boundaries.
- **Theme divergence:** 11ty and MUI could diverge visually -> enforce shared tokens as source of truth.
- **SWA config mismatch:** wrong `app_location`/`output_location` breaks deploys -> standardise build paths per app and document in workflows.
- **Domain confusion:** users mixing `www` and `app` expectations -> keep navigation boundaries explicit and avoid auth surfaces on marketing.

Comparative Analysis Matrix (Starter & Deployment)

Option	Maintainability	Trust/Privacy Fit	Delivery Speed	Complexity	Score
Chosen: Official Vite + Functions + 11ty, pnpm, 2x SWA	5	5	4	4	**18**
Single SWA for app + marketing	3	3	4	3	13
MUI-opinionated starter	3	4	4	3	14
npm workspaces instead of pnpm	4	5	4	4	17

Pre-mortem (What Could Go Wrong)

- Build pipelines drift between apps -> lock shared tooling and document per-app build outputs.
- Theme package becomes stale -> treat tokens as source of truth and update in one place only.
- Marketing deploy breaks due to 11ty output path mismatch -> standardise output dir and wire in SWA config.
- App SWA fails to route auth correctly -> ensure `staticwebapp.config.json` lands in app build output.

Clarity Checks (Rubber Duck)

- **Simple:** We will create three apps in one repo, share a theme, and deploy two SWAs (app+API, marketing).
- **Detailed:** `/apps/web` (Vite+React+MUI+Storybook), `/apps/api` (Functions .NET isolated), `/apps/marketing` (11ty), `/packages/theme` for shared tokens.
- **Technical:** pnpm workspaces manage dependencies; SWA workflows point at app and api folders; 11ty output path aligns with marketing SWA.

Red vs Blue (Starter & Deployment)

- **Red Team:** two SWAs and a monorepo increase config surface area -> higher chance of mis-deploys.

- **Blue Team:** standardised app locations and documented build outputs mitigate; CI checks can validate config.
- **Red Team:** shared theme package could create coupling across apps -> harder to evolve independently.
- **Blue Team:** keep tokens minimal and versioned; avoid deep UI dependencies across app and marketing.

Note: Project initialisation using these commands should be the first implementation story.

Core Architectural Decisions

Data Architecture

Decision Priority: Critical (blocks implementation)

Database Choice: Azure Table Storage (Azure.Data.Tables SDK, latest verified 12.11.0). **Rationale:** Aligns with MVP constraints, cost, and simplicity.

Data Modelling Approach (Table per entity):

- **Memberships**: `PartitionKey = ProgrammeId`, `RowKey = UserId`
- **Cycles**: `PartitionKey = ProgrammeId`, `RowKey = CycleDate` (ISO, programme TZ)
- **Matches**: `PartitionKey = ProgrammeId`, `RowKey = MatchId` (deterministic from ProgrammeId + CycleDate + sorted UserIds)
- **Events (optional MVP)**: `PartitionKey = ProgrammeId`, `RowKey = Timestamp + Guid` (only if needed)

Validation Strategy: Separate schemas per layer.

- Web: Zod validation.
- API: explicit validation at Functions boundaries.
- No shared schema package in MVP.

Schema Evolution / Migrations: Additive changes only; no migrations in MVP.

Caching: None in MVP (prioritise observability and simplicity).

Implications / Cascading Decisions:

- Deterministic MatchId requires stable canonicalisation of participant IDs.
- CycleDate timezone handling must align with programme TZ decision.
- Event table is optional; if deferred, logging must still satisfy audit requirements.

Authentication & Security

Decision Priority: Critical (blocks implementation)

Authorisation Pattern: Role-based, enforced server-side per route.

- Roles: Participant, Programme Owner, Admin.
- Executive Sponsor has no separate role in MVP (same visibility as Programme Owner).
- Role enforcement via `staticwebapp.config.json` and API checks.

Identity & Claims Handling:

- EasyAuth for identity; validate claims on every request, deny-by-default.
- Never trust client-provided identity; decode SWA client principal header as untrusted input.

Encryption & Secrets:

- Azure Storage encryption at rest by default; rely on it for Table Storage data.
- No Key Vault in MVP; use SWA/Functions app settings (environment variables) and GitHub Secrets for deployment.
- **Upgrade hook:** SecretsProvider abstraction so Key Vault can be added later without refactoring callers.

Role Handling (All-Free constraint):

- Keep the concept of roles in API logic, but implement minimal role handling:
  - Default everyone to Participant.
  - Programme Owner/Admin via small allowlist (Entra Object IDs) in Table Storage or app settings.
  - No dynamic role management UI in MVP.
- No manager visibility remains non-negotiable.
- **Upgrade hook:** RolesProvider abstraction to support dynamic role management later.

API Security:

- Explicit allowlist per route.
- Reject missing/malformed claims; return 401/403.
- No custom auth flows.

Audit / Event Logging:

- Immutable system-event log only (no user behaviour analytics).
- Log: matching run start/complete, notification attempts/failures, admin config changes.
- Include correlation IDs for debugging.

Implications / Cascading Decisions:

- Role allowlist source must be documented (Table vs app settings) and secured.

API & Communication Patterns

Decision Priority: Critical (blocks implementation)

API Design: RESTful JSON, noun-based routes.

- Keep route surface small and predictable.
- DTOs only at the boundary.

API Documentation: OpenAPI.NET generated specs as the source of truth.

- Must stay aligned with DTOs and contract tests.
- Use Microsoft.OpenApi (latest verified 3.1.2).

Error Handling Standard: Problem Details JSON (RFC 9457, obsoletes 7807).

- Same shape everywhere.
- Human-readable `title` and `detail`.
- Correlation ID included for support/debug.

Rate Limiting: None in MVP.

- If abuse appears, add a basic per-user throttle later.

Implications / Cascading Decisions:

- Standardise error envelope and ensure correlation ID is generated per request.
- OpenAPI specs must be regenerated when DTOs change.

Frontend Architecture

Decision Priority: Important (shapes implementation)

State Management: React hooks + Context only. No global state library in MVP.

Routing: React Router v6 with minimal route guards.

- Guards only for auth-required vs public routes.
- Clean redirect back to intended page after EasyAuth.
- No client-side permission logic beyond auth presence (server remains authoritative).

Forms: Native forms + small helpers only. No React Hook Form.

Performance: Keep simple; no explicit code-splitting beyond Vite defaults.

Implications / Cascading Decisions:

- Auth guard UX must avoid loops and preserve return URLs safely.
- Any future state library must justify added complexity.

Infrastructure & Deployment

Decision Priority: Important (shapes implementation)

CI/CD: Standard Azure Static Web Apps GitHub Actions workflows.

- One workflow for app+API SWA.
- One workflow for marketing SWA.

Environment Configuration:

- `.env` for local dev.
- SWA app settings for deploy-time config.
- No Key Vault in MVP; use app settings and GitHub Secrets.
- **Upgrade hook:** SecretsProvider abstraction to enable Key Vault later.

Monitoring:

- No Application Insights in MVP; basic platform logs only.

- Log errors and matching-run outcomes only.
- **Upgrade hook:** TelemetryProvider abstraction so App Insights can be enabled later with capped logging.
- Minimal alerting if enabled later: matching run failures, notification send failures, auth failures spike, storage errors.

Scaling:

- Default consumption/serverless only; no pre-provisioned scaling controls for MVP.

Implications / Cascading Decisions:

- Alert thresholds and routing must be defined (team email/Slack).
- App settings must be documented per SWA (app vs marketing).

Implementation Patterns & Consistency Rules

Pattern Categories Defined

Critical Conflict Points Identified: Naming, file structure, API formats, date handling, error/loading patterns.

Naming Patterns

Database Naming Conventions:

- **Tables:** PascalCase (`Memberships`, `Cycles`, `Matches`, `Events`)
- **Entity properties:** camelCase (`programmeId`, `cycleDate`, `createdAt`)

API Naming Conventions:

- **Routes:** plural, noun-based (`/programmes`, `/matches`, `/me`)
- **Route params:** `{id}` (`/programmes/{programmeId}`)
- **JSON fields:** camelCase (`matchedUserId`, `cycleDate`)

Code Naming Conventions:

- **Components:** PascalCase (`MatchCard`)
- **Frontend files:** kebab-case (`match-card.tsx`)
- **Functions/vars:** camelCase (`getProgrammeById`, `userId`)

Structure Patterns

Project Organization:

- Components organised **by feature** (`/features/...`)
- Shared utilities in `/lib`

Tests:

- Co-located `*.test.ts` / `*.test.tsx` alongside source files

Format Patterns

API Response Formats:

- **Direct payloads** only (no `{ data, error }` wrapper)
- **Problem Details** is the only standard error wrapper shape

Data Exchange Formats:

- JSON fields: camelCase
- Dates: ISO 8601 strings with timezone offsets where relevant
    - Cycle date uses programme TZ

Communication Patterns

Events (if/when event log used):

- Event rows use programme-scoped PartitionKey
- Event payload fields in camelCase

Process Patterns

Error Handling Patterns:

- Global error boundary for unexpected UI crashes
- Per-call handling for API errors (show inline error states)

Loading State Patterns:

- Local component state only (no shared loading context)

Pattern Risks

- **Route naming drift:** agents might reintroduce singular routes -> enforce in code review checklist.
- **Date format drift:** ISO 8601 vs epoch -> contract tests must assert ISO 8601.
- **Error shape drift:** problem-details vs ad-hoc error objects -> lint/test for error envelope consistency.
- **File naming drift:** PascalCase vs kebab-case on frontend -> enforce via linting rule.

First Principles (Why These Patterns Exist)

- **Consistency beats cleverness:** shared naming/format rules prevent invisible integration bugs.
- **Directness over abstraction:** direct payloads and simple routing reduce accidental divergence.
- **Trust is the product:** error handling and date formats must be predictable and boring.

Failure Modes

- **Mixed casing in routes/fields:** breaks client/server contract silently.
- **Inconsistent date formats:** invalid comparisons and broken sorting.
- **Non-standard error shapes:** UI error handling fails unpredictably.
- **Misplaced tests:** reduced coverage on contract boundaries.

Consistency Checks

- Routes are plural and noun-based; params always `{id}`.

- JSON fields and entity properties are camelCase everywhere.
- Frontend file names are kebab-case; component names are PascalCase.
- Problem Details is the only error wrapper; success responses are direct payloads.

Enforcement Guidelines

All AI Agents MUST:

- Use PascalCase table names and camelCase for entity/JSON fields.
- Use plural, noun-based API routes with `{id}` params.
- Keep frontend files kebab-case and components PascalCase.
- Use ISO 8601 date strings; Problem Details only for errors.

Pattern Enforcement:

- Add contract tests for ISO 8601 dates and Problem Details error shape.
- Review checklist includes naming/format rules; deviations require explicit approval.

Pattern Examples (Condensed)

Good:

- `GET /programmes/{programmeId}` -> `{ programmeId, cadence, cycleDate }`
- `features/matches/match-card.tsx` exports `MatchCard`

Anti-Patterns:

- `/programme/123`
- `match_card.tsx`
- `{ data: {...} }` wrapper for success
- epoch timestamps

Project Structure & Boundaries

Complete Project Directory Structure

```
collabolatte/
├── README.md
├── pnpm-workspace.yaml
├── package.json
├── .gitignore
├── .env.example
├── .github/
│   └── workflows/
│       ├── swa-app.yml           # app+api SWA
│       └── swa-marketing.yml     # marketing SWA
├── apps/
│   ├── web/
│   │   ├── package.json
│   │   ├── vite.config.ts
│   │   ├── tsconfig.json
```

```
│   │   ├── staticwebapp.config.json
│   │   ├── src/
│   │   │   ├── app.tsx
│   │   │   ├── main.tsx
│   │   │   ├── routes/
│   │   │   ├── features/
│   │   │   │   ├── account/          # includes transparency page
│   │   │   │   ├── programmes/
│   │   │   │   ├── matches/
│   │   │   │   └── admin/
│   │   │   ├── lib/
│   │   │   │   ├── api/
│   │   │   │   ├── auth/
│   │   │   │   └── utils/
│   │   │   ├── components/
│   │   │   └── styles/
│   │   └── tests/                # co-located tests under feature folders
│   ├── api/
│   │   ├── collabolatte-api.sln
│   │   ├── src/
│   │   │   ├── Collabolatte.Api/
│   │   │   │   ├── Program.cs
│   │   │   │   ├── Functions/
│   │   │   │   │   ├── Programmes/
│   │   │   │   │   ├── Matches/
│   │   │   │   │   ├── Participation/
│   │   │   │   │   └── Admin/
│   │   │   │   ├── Contracts/
│   │   │   │   ├── Services/
│   │   │   │   ├── Repositories/
│   │   │   │   ├── Models/
│   │   │   │   ├── Validation/
│   │   │   │   └── Logging/
│   │   │   └── Collabolatte.Api.Tests/
│   │   │       ├── Functions/        # contract tests live here
│   │   │       ├── Services/
│   │   │       └── Repositories/
│   │   └── host.json
│   └── marketing/
│       ├── package.json
│       ├── .eleventy.js
│       ├── src/
│       │   ├── _data/
│       │   ├── _includes/
│       │   ├── pages/
│       │   ├── assets/
│       │   └── styles/
│       └── dist/                  # 11ty output (SWA output_location)
├── packages/
│   └── theme/
│       ├── package.json
│       ├── tokens.css
│       └── muiTheme.ts
└── tests/
```

```
    └── e2e/
        ├── playwright.config.ts
        └── specs/
```

Architectural Boundaries

API Boundaries:

- REST endpoints exposed in `apps/api/src/Collabolatte.Api/Functions/*`
- Auth boundary enforced per Function (EasyAuth claims validation)

Component Boundaries:

- Frontend features isolated under `apps/web/src/features/*`
- Shared UI in `apps/web/src/components`
- Shared utilities in `apps/web/src/lib`

Service Boundaries:

- Domain services in `apps/api/src/Collabolatte.Api/Services`
- Data access via `apps/api/src/Collabolatte.Api/Repositories`

Data Boundaries:

- Table Storage entities under `apps/api/src/Collabolatte.Api/Models`
- Partition rules enforced in repositories

Requirements to Structure Mapping

Feature Mapping (from FR categories):

- **Identity & Access:** `apps/web/src/features/account`, `apps/api/src/Collabolatte.Api/Functions/Account`
- **Programme Participation:** `apps/web/src/features/programmes`, `apps/api/src/Collabolatte.Api/Functions/Participation`
- **Matching:** `apps/web/src/features/matches`, `apps/api/src/Collabolatte.Api/Functions/Matches`
- **Notifications:** `apps/api/src/Collabolatte.Api/Services/Notifications`
- **Administration:** `apps/web/src/features/admin`, `apps/api/src/Collabolatte.Api/Functions/Admin`
- **Transparency Page:** `apps/web/src/features/account`

Cross-Cutting Concerns:

- Auth/claims validation: `apps/api/src/Collabolatte.Api/Validation`
- Logging/event records: `apps/api/src/Collabolatte.Api/Logging`

Integration Points

Internal Communication:

- Web app calls API endpoints under `/api/*`.
- API reads/writes Table Storage and sends email via ACS.

External Integrations:

- EasyAuth (SWA) for identity.
- Azure Communication Services for email delivery.

Data Flow:

- Client -> API -> Table Storage
- Scheduler -> Matching service -> Notifications

File Organization Patterns

Configuration Files:

- `staticwebapp.config.json` must be emitted into the web build output root.
- `.env.example` at repo root.

Source Organization:

- Feature-first in web; feature-grouped Functions in API.

Test Organization:

- Co-located tests in web and API.
- Contract tests under `apps/api/src/Collabolatte.Api.Tests/Functions/`.
- Playwright E2E in `/tests/e2e`.

Asset Organization:

- Web assets in `apps/web/src/styles`
- Marketing assets in `apps/marketing/src/assets`

Development Workflow Integration

Development Server Structure:

- `apps/web` runs Vite dev server.
- `apps/api` runs Functions host locally.
- `apps/marketing` runs 11ty build/watch.

Build Process Structure:

- App SWA builds `apps/web` with API path `apps/api`.
- Marketing SWA builds `apps/marketing` output to `dist`.

Deployment Structure:

- `app.collabolatte.co.uk` SWA: `app_location=apps/web`, `api_location=apps/api`.
- `www.collabolatte.co.uk` SWA: `app_location=apps/marketing`, `output_location=dist`.

Structure Dependencies

- `/packages/theme` is a shared contract: both `apps/web` and `apps/marketing` depend on it for visual consistency.
- `apps/web` depends on `apps/api` endpoint shapes; API changes must update OpenAPI + contract tests.
- `tests/e2e` depends on stable routes and data fixtures; keep environment config consistent across SWA and local.

Cross-Functional Structure Trade-offs

- **PM:** wants clear ownership boundaries -> feature-first layout keeps scope visible.
- **Engineering:** wants low merge conflicts -> shared `/packages/theme` reduces duplicated styling.
- **UX:** wants brand consistency across app and marketing -> tokens.css + MUI theme in one place.
- **Privacy/Legal:** wants clear separation between marketing and authenticated app -> distinct apps and SWA workflows.

Structure Risks

- **Theme package drift:** shared tokens not updated across apps -> add a simple version bump/checklist.
- **API feature sprawl:** Functions folders diverge from routes -> enforce mapping in OpenAPI + tests.
- **Marketing build output mismatch:** 11ty output path doesn't match SWA config -> keep `dist/` consistent and documented.

Architecture Validation Results

Coherence Validation ☑

Decision Compatibility: All core choices are compatible: React/Vite + SWA EasyAuth + Functions (.NET isolated) + Table Storage + pnpm monorepo. The All-Free constraint is reflected in secrets/roles/monitoring choices.

Pattern Consistency: Naming, routing, file structure, and error patterns align with the tech stack and API decisions. Problem Details is the only error wrapper; success responses are direct.

Structure Alignment: Project structure supports the decisions (feature-first UI, feature-grouped Functions, shared theme package, co-located tests + E2E).

Requirements Coverage Validation ☑

Epic/Feature Coverage: FR categories map cleanly to `apps/web/src/features/*` and `apps/api/src/Collabolatte.Api/Functions/*`.

Functional Requirements Coverage: All FR categories (identity, participation, matching, notifications, admin, privacy) are supported by current architecture.

Non-Functional Requirements Coverage: Security, privacy, and trust constraints are reflected in auth patterns, data minimisation, and logging rules. Availability/performance constraints are consistent with serverless + simple UI.

Implementation Readiness Validation ☑

Decision Completeness: Critical decisions documented (data, auth, API, frontend, infra). Versions captured where relevant.

Structure Completeness: Directory tree is specific and complete across web, API, marketing, theme, and tests.

Pattern Completeness: Conflict points are addressed with concrete conventions and enforcement rules.

Gap Analysis Results

Important Gap (Resolved):

- **Role allowlist source:** Table Storage is the source of truth (auditable, changeable without redeploys).

Nice-to-Have:

- Correlation ID format defined as GUID for logs and Problem Details.

Validation Issues Addressed

- Role allowlist source resolved: **Table Storage**.
- Correlation ID format locked: **GUID**.
- Contract tests for ISO 8601 dates and Problem Details are mandatory gates.

Architecture Completeness Checklist

☑ Requirements Analysis

- ☑ Project context thoroughly analysed
- ☑ Scale and complexity assessed
- ☑ Technical constraints identified
- ☑ Cross-cutting concerns mapped

☑ Architectural Decisions

- ☑ Critical decisions documented with versions
- ☑ Technology stack fully specified
- ☑ Integration patterns defined
- ☑ Performance considerations addressed

☑ Implementation Patterns

- ☑ Naming conventions established
- ☑ Structure patterns defined
- ☑ Communication patterns specified
- ☑ Process patterns documented

☑ Project Structure

- ☑ Complete directory structure defined
- ☑ Component boundaries established
- ☑ Integration points mapped

- ☑ Requirements to structure mapping complete

Architecture Readiness Assessment

Overall Status: READY FOR IMPLEMENTATION
Confidence Level: High

Key Strengths:

- Trust-first architecture with clear boundaries
- Minimal, consistent conventions to prevent agent drift
- Cost-aware deployment plan with upgrade hooks

Areas for Future Enhancement:

- Optional Key Vault / App Insights enablement
- Role management UI (post-MVP)

Implementation Handoff

AI Agent Guidelines:

- Follow all architectural decisions exactly as documented
- Use implementation patterns consistently across all components
- Respect project structure and boundaries
- Refer to this document for all architectural questions

First Implementation Priority:

- Initialise monorepo structure + scaffolding per starter decisions

Architecture Completion Summary

Workflow Completion

Architecture Decision Workflow: COMPLETED ☑
Total Steps Completed: 8
Date Completed: 2026-01-14
Document Location: _bmad-output/planning-artifacts/architecture.md

Final Architecture Deliverables

📋 Complete Architecture Document

- All architectural decisions documented with specific versions where required
- Implementation patterns ensuring AI agent consistency
- Complete project structure with all files and directories
- Requirements-to-structure mapping
- Validation confirming coherence and completeness

🏗️ Implementation Ready Foundation

- 5 core architectural decision categories finalised
- 14 consistency and enforcement patterns defined
- 5 architectural components specified (web, api, marketing, theme, e2e)
- 8 functional requirement categories fully supported

AI Agent Implementation Guide

- Technology stack with verified versions
- Consistency rules that prevent implementation conflicts
- Project structure with clear boundaries
- Integration patterns and communication standards

Implementation Handoff

For AI Agents: This architecture document is your complete guide for implementing collabolatte. Follow all decisions, patterns, and structures exactly as documented.

First Implementation Priority: Initialise the monorepo structure + scaffolding per starter decisions.

Development Sequence:

1. Initialise project using documented starter template
2. Set up development environment per architecture
3. Implement core architectural foundations
4. Build features following established patterns
5. Maintain consistency with documented rules

Quality Assurance Checklist

Architecture Coherence

- ☑ All decisions work together without conflicts
- ☑ Technology choices are compatible
- ☑ Patterns support the architectural decisions
- ☑ Structure aligns with all choices

Requirements Coverage

- ☑ All functional requirements are supported
- ☑ All non-functional requirements are addressed
- ☑ Cross-cutting concerns are handled
- ☑ Integration points are defined

Implementation Readiness

- ☑ Decisions are specific and actionable
- ☑ Patterns prevent agent conflicts
- ☑ Structure is complete and unambiguous
- ☑ Examples are provided for clarity

Project Success Factors

🎯 Clear Decision Framework
Every technology choice was made collaboratively with clear rationale, ensuring all stakeholders understand the architectural direction.

🔧 Consistency Guarantee
Implementation patterns and rules ensure that multiple AI agents will produce compatible, consistent code that works together seamlessly.

📋 Complete Coverage
All project requirements are architecturally supported, with clear mapping from business needs to technical implementation.

🏗️ Solid Foundation
The chosen starter template and architectural patterns provide a production-ready foundation following current best practices.

Architecture Status: READY FOR IMPLEMENTATION ☑
Next Phase: Begin implementation using the architectural decisions and patterns documented herein.
Document Maintenance: Update this architecture when major technical decisions are made during implementation.